Characterizing Secure Dynamic Web Applications Scalability

Jordi Guitart, Vicenç Beltran, David Carrera, Jordi Torres and Eduard Ayguadé
European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department - Technical University of Catalonia
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034 Barcelona (Spain)
{jguitart, vbeltran, dcarrera, torres, eduard}@ac.upc.edu

Abstract

Security in the access to web contents and the interaction with web sites is becoming one of the most important issues in Internet. Servers need to provide certain levels of security so that the user feels comfortable when running the applications that provide the services he/she requires. HTTP over SSL is the most used solution, providing mutual authentication between the two interacting parts. The SSL protocol does not introduce complexity in web applications but increases the computational demand on the server, reducing its capacity to serve large number of clients and increasing the time to serve them.

In order to compensate the degradation in the quality of service, the server needs to be upgraded with additional resources, mainly processors and memory. In this paper we analyze the scalability of servers that run secure dynamic web applications. We analyze how the server behaves when it is stressed with different number of clients and how the quality of service is degraded. We perform a detailed analysis of the server behavior and analyze the impact of adding more processors to the system that runs the server. The analysis is done using a fine-grained analysis framework that considers all levels in the application server execution (i.e. application, server, JVM and OS kernel). The RUBiS auction site benchmark is used to stress a Tomcat application server running on a commodity 4-way multiprocessor Intel platform with Linux.

1. Introduction

Current web sites have to face two issues that affect directly to the site scalability. First, the web community is growing day after day, increasing exponentially the load that sites must support to satisfy all clients requests. Second, dynamic web content is becoming popular on current sites. At the same time, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. Security between network nodes over the Internet is traditionally provided using HTTPS [32]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [15]), you can perform mutual authentication of both the sender and receiver of messages and ensure message confidentiality. This process involves X.509 certificates that are configured on both sides of the connection. This widespread diffusion of dynamic web content and SSL increases the performance demand on application servers that host the sites. Due to these two facts, the scalability of these application servers has become a crucial issue in order to support the maximum number of concurrent clients demanding secure dynamic web content.

Characterizing application servers scalability is something more complex than measuring the application server performance with different number of clients and determining the load that saturates the server. A complete characterization must also supply the causes of this saturation, giving to the server administrator the chance and the information to improve the server scalability by avoiding its saturation. For this reason, this characterization requires of powerful analysis tools that allow an in-depth analysis of the application server behavior and its interaction with the other system elements (including distributed clients, a database server, etc.). These tools must support and consider all the levels involved in the execution of web applications (operating system, Java Virtual Machine, application server and application) if they want to provide significant performance information to the administrators because the origin of performance problems can reside in any of these levels or in their interaction.

A complete scalability characterization must also consider another important issue: the scalability relative to the resources. The analysis for determining the causes of server saturation can reveal that some resource is being a bottleneck for server scalability. In this case, a good option could be the addition of more resources of this type and the evaluation of the effect of this addition on server behavior in order to determine the causes of server saturation. On the other side, although any resource has been detected as a bottleneck for server scalability, the analysis of server behavior when adding more resources can be performed to verify if server saturation problem remains unresolved.

In this paper we present a characterization of secure dynamic web applications scalability divided in two parts. First, we measure the vertical scalability of the server if running with different number of processors, determining the impact of adding more processors on server saturation. Second, we perform a detailed analysis of the server behavior using a performance analysis framework, in order to determine the causes of the server saturation when running with different number of processors. This framework considers all levels involved in the application server execution, allowing a fine-grain analysis of dynamic web applications.

The rest of the paper is organized as follows: Section 2 introduces dynamic web applications. Section 3 introduces the SSL protocol used to provide security capabilities when accessing web content. Section 4 describes our proposal for analyzing the scalability of secure dynamic web applications. Section 5 describes the experimental environment used in our evaluation. Section 6 presents our evaluation of secure dynamic web applications scalability. Section 7 presents the related work and finally, Section 8 presents the conclusions of this paper.

2. Dynamic web applications

Dynamic web applications are a case of multi-tier application and are mainly composed of a Client tier and a Server tier, which in its turn uses to consist of a front-end web server, an application server and a back-end database. The client tier is responsible of interacting with application users and to generate requests to be attended by the server. The server tier implements the logic of the application and is responsible of serving user-generated requests.

When the client sends to the web server an HTTP request for dynamic content, the web server forwards the request to the application server (as understood in this paper, a web server only serves static content), which is the dynamic content server. The application server executes the corresponding code, which may need to access the database to generate the response. The application server formats and assembles the results into an HTML page, which is returned as an HTTP response to the client.

The implementation of the application logic in the application server may take various forms, including PHP [31], Microsoft Active Server Pages [26], Java Servlets [34] and Enterprise Java Beans (EJB) [33]. This study focuses on Java Servlets, but the same methodology can be applied with the other mechanisms for generating dynamic web content, in order to characterize their scalability.

A servlet is a Java class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

Servlets access the database explicitly, using the standard JDBC interface, which is supported by all major databases. Servlets can use all the features of Java. In particular, they can use Java built-in synchronization mechanisms to perform locking operations.

3. SSL protocol

The SSL protocol provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. To obtain these objectives it uses a combination of public-key and private-key cryptography algorithm and digital certificates (X.509).

The SSL protocol does not introduce a new degree of complexity in web applications structure because it works almost transparently on top of the socket layer. However, SSL increases the computation time necessary to serve a connection remarkably, due to the use of cryptography to achieve their objectives. This increment has a noticeable impact on server performance, which can be appreciated on Figure 1. This figure compares the throughput obtained by the Tomcat application server, configured as described in Section 5, using secure connections versus using normal connections. Notice that the maximum throughput

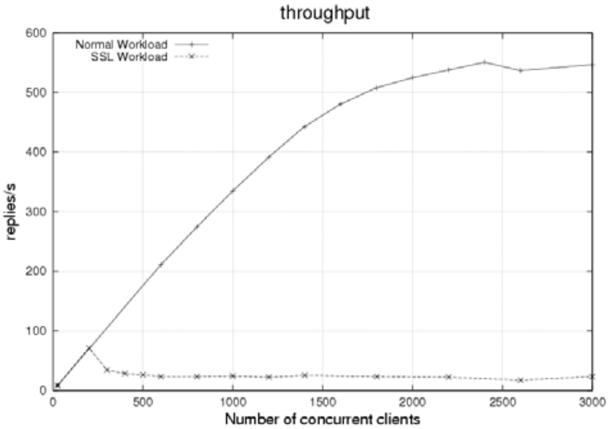

Figure 1. Tomcat scalability when serving secure vs. non-secure connections

obtained when using SSL connections is 72 replies/s and the server scales only until 200 clients. On the other side, when using normal connections the throughput is considerably higher (550 replies/s) and the server can scale until 1700 clients. Finally, notice also that when the server is saturated, if attending normal connections, the server can maintain the throughput if new clients arrive, but if attending SSL connections, the server cannot maintain the throughput and the performance is degraded. More information about the impact of using SSL on server performance can be found on [9].

The SSL protocol fundamentally has two phases of operation: SSL handshake and SSL record protocol. We will do an overview of the SSL handshake phase, which is the responsible of most of the computation time required when using SSL. The detailed description of the whole protocol can be found in RFC 2246 [14].

The SSL handshake allows the server to authenticate itself to the client using public-key techniques like RSA, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server.

Two different SSL handshake types can be distinguished: The full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake. This negotiation includes parts that spend a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine to be around 175 ms.

The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server but using an existing SSL connection. As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, reducing considerably the computation demand for performing a resumed SSL handshake. We have measured the computational demand of a resumed handshake in a 1.4 GHz Xeon machine to be around 2 ms. Notice the big difference between negotiate a full SSL handshake respect to negotiate a resumed SSL handshake (175 ms versus 2 ms).

4. Servers scalability

The scalability of an application server is defined as the ability to maintain a site availability, reliability, and performance as the amount of simultaneous web traffic, or load, hitting the application server increases [19].

Given this definition, the scalability of an application server can be represented measuring the performance of the application server while the load increases. With this representation, the load that provokes the saturation of the server can be detected. We consider that the application server is saturated when it is unable to maintain the site availability, reliability, and performance (i.e. the server does not scale). As derived from the definition, when the server is saturated, the performance is degraded (lower throughput and higher response time) and the number of client requests refused is increased.

At this point, two questions should appear to the reader (and of course, to the application server administrator). First, the load that provokes the saturation of the server has been detected, but why is this load causing the server performance to degrade? In other words, in which parts of the system (CPU, database, network, etc.) will a request be spending most of its execution time at the saturation points? In order to answer this question, we propose to analyze the application server behavior using a performance analysis framework, which considers all levels involved in the application server execution (operating system, JVM, application server and application), allowing a fine-grain analysis of dynamic web applications.

Second, the application server scalability with given resources has been measured, but how would affect to the application server scalability the addition of more resources? This adds a new dimension to the application servers scalability: the measurement of the scalability relative to the resources. This scalability can be done in two different ways: vertical and horizontal.

Vertical scalability (also called scaling up) is achieved by adding capacity (memory, processors, etc.) to an existing application server and requires few to no changes to the architecture of the system. Vertical scalability increases the performance (in theory) and the manageability of the system, but decreases the reliability and availability (single failure is more likely to lead to system failure). We will consider this kind of scalability relative to the resources in this paper.

Horizontal scalability (also called scaling out) is achieved by adding new application servers to the system, increasing the complexity of the system. Horizontal scalability increases the reliability, the availability and the performance (depends on load balancing), but decreases the manageability (there are more elements in the system).

The analysis of the application server behavior will provide us with hints to answer the question about how would affect to the application server scalability the addition of more resources. If we detect that some resource is being a bottleneck for the application server performance, this encourages the addition of new resources of this type (vertical scaling), the measurement of the scalability with this new configuration and the analysis of the application server behavior with the performance analysis framework to determine the improvement on the server scalability and the new causes of server saturation.

On the other side, if we upgrade a resource that is not being a bottleneck for the application server performance, we can verify with the performance analysis framework that scalability is not improved and the causes of server performance degradation remain unresolved. This observation justifies why with vertically scaling performance is improved only in theory, depending if the added resource is a bottleneck for server performance or not. This observation also motivates the analysis of the application server behavior in order to detect the causes of saturation before adding new resources.

5. Experimental environment

5.1 Tomcat servlet container

We use Tomcat v5.0.19 [22] as the application server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container too. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this paper we use Tomcat as a standalone server.

Tomcat follows a connection service schema where, at a given time, one thread (an HttpProcessor) is responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this HttpProcessor will be responsible of attending and serving the received requests through the persistent connection established with the client, while another HttpProcessor will continue accepting new connections. HttpProcessors are commonly chosen from a pool of threads in order to avoid thread creation overheads.

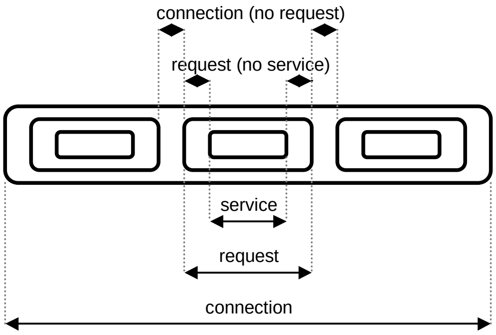

Figure 2. Tomcat persistent connection pattern

Persistent connections are a feature of HTTP 1.1 that allows serving different requests using the same connection, saving a lot of work and time for the web server, client and the network, considering that establishing and tearing down HTTP connections is an expensive operation.

The pattern of a persistent connection in Tomcat is shown in Figure 2. In this example, three different requests are served through the same connection. The rest of the time (*connection (no request)*) the server is maintaining opened the connection waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. Notice that within every *request* is distinguished the *service* (execution of the servlet implementing the demanded request) from the *request (no service)*. This is the pre and post process that Tomcat requires to invoke the servlet that implements the demanded request.

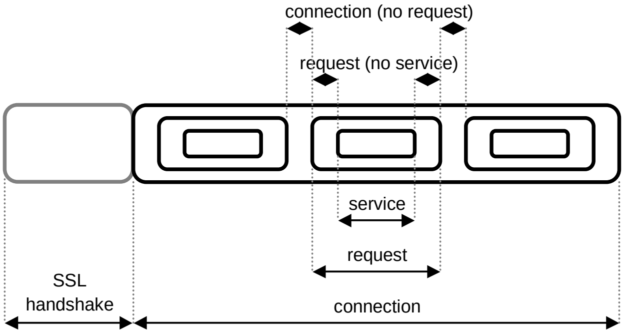

Figure 3. Tomcat secure persistent connection pattern

Figure 3 shows the pattern of a secure persistent connection in Tomcat. Notice that when using SSL the pattern of the HTTP persistent connection is maintained, but the underlying SSL connection supporting this persistent HTTP connection must be established previously, negotiating a SSL handshake, which can be full or resumed depending if a SSL Session ID is reused. For instance, if a client must establish a new HTTP connection because its current HTTP connection has been closed by the server due to connection persistence timeout expiration, as it reuses the underlying SSL connection, it negotiates a resumed SSL handshake.

We have configured Tomcat setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 seconds.

5.2 Auction site benchmark (RUBiS)

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [1] benchmark servlets version 1.4 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [27]. This tool, which support both HTTP and HTTPS protocols, allows the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented in this paper were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of concurrent clients interacting with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and follows a link embedded in the response. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. This emulates the “thinking” period of a real client who takes a period of time before clicking on the next request. The think time is generated from a negative exponential distribution with a mean of 7 seconds. Httperf allows also configuring a client timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated. We have configured Httperf setting the client timeout value to 10 seconds. RUBiS defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions.

5.3 Performance analysis framework

In order to determine the causes of the server saturation, we propose to analyze the application server behavior using a performance analysis framework. This framework, which consists of an instrumentation tool called Java Instrumentation Suite (JIS [10]) and a visualization and analysis tool called Paraver [30], considers all levels involved in the application server execution (operating system, JVM, application server and application), allowing a fine-grain analysis of dynamic web applications. For example, the framework can provide detailed information about thread status, system calls (I/O, sockets, memory & thread management, etc.), monitors, services, connections, etc. Further information about the implementation of the performance analysis framework and its use for the analysis of dynamic web applications can be found in [10] and [17].

5.4 Hardware & software platform

Tomcat runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [29] as our database server with the MM.MySQL v3.0.8 JDBC driver. MySQL runs on a 2-way Intel XEON 2.4 GHz with 2 GB RAM. We have also a 2-way Intel XEON 2.4 GHz with 2 GB RAM machine running the workload generator (Httperf 0.8). Each client emulation machine emulates the configured number of clients performing requests to the server during 10 minutes using the browsing mix (read-only interactions). All the machines run the 2.6.2 Linux kernel. Server machine is connected with client machine through a 1 Gbps Ethernet interface. Database and server machine are direct connected through 100 Mbps fast Ethernet crossed-link. For our experiments we use the Sun JVM 1.4.2 for Linux, using the server JVM instead of the client JVM and setting the initial and the maximum Java heap size to 1024 MB, which we have proven to be enough to avoid memory being a bottleneck for performance.

All the tests are performed with the common RSA-3DES-SHA cipher suit. Handshake is performed with 1024 bit RSA key. Record protocol uses triple DES to encrypt all application data. Finally, SHA digest algorithm provides the Message Authentication Code (MAC).

6. Evaluation

In this section we present the scalability characterization of Tomcat application server when running the RUBiS benchmark using SSL. The evaluation is divided in two parts. First, we evaluate the vertical scalability of the server when running with different number of processors, determining the impact of adding more processors on server saturation (can the server support more clients before saturating?) Second, we perform a detailed analysis of the server behavior using a performance analysis framework, in order to determine the causes of the server saturation when running with different number of processors.

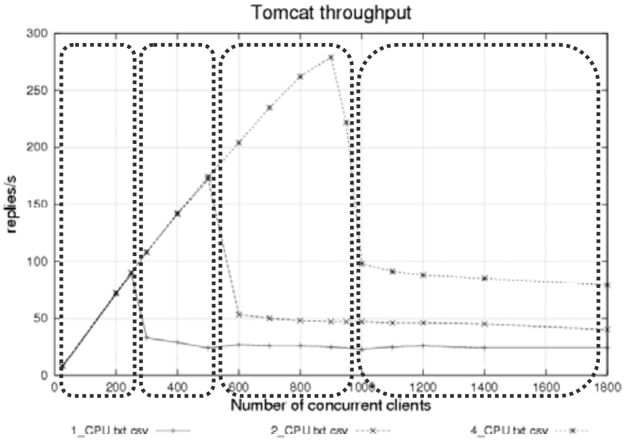

Figure 4. Tomcat scalability with different number of processors

6.1 Tomcat vertical scalability

Figure 4 shows the Tomcat scalability when running with different number of processors, representing the server throughput as a function of the number of clients. Notice that for a given number of processors, the server throughput increases linearly with respect to the input load (the server scales) until a determined number of clients hit the server. At this point, the throughput achieves its maximum value. Table 1 shows the number of clients that saturate the server and the maximum achieved throughput before saturating when running with one, two and four processors. Notice that running with more processors allows the server to handle more clients before saturating, so the maximum achieved throughput is higher.

Notice also that the same throughput can be achieved, as shown in Figure 1, with a single processor when SSL is not used. This means that when using secure connections, the computing capacity provided when adding more processors is spent on supporting the SSL protocol.

Table 1. Number of clients that saturate the server and maximum achieved throughput before saturating

number of processors	number of clients	throughput (replies/s)
1	250	90
2	500	172
4	950	279

When the number of clients that saturate the server has been achieved, the server throughput degrades to approximately the 30% of the maximum achievable throughput, as shown in Table 2. This table shows the average throughput obtained when the server is saturated when running with one, two and four processors. Notice that, although the throughput obtained has been degraded in all cases when the server has reached a saturated state, running with more processors improves the throughput (duplicating the number of processors, the throughput almost duplicates too).

Table 2. Average server throughput when saturated

number of processors	throughput (replies/s)
1	25
2	50
4	90

6.2 Tomcat scalability analysis

In order to perform a detailed analysis of the server, we have selected four different loads: 200, 400, 800 and 1400 clients, each one corresponding to one of the zones observed in Figure 4. These zones group the loads with similar behavior of the server. In order to conduct this analysis, we use the performance analysis framework described in Section 5.3.

The analysis methodology consists of comparing the server behavior when it is saturated (400 clients when running with one processor, 800 clients when running with two processors and 1400 clients when running with four processors) with when it is not (200 clients when running with one processor, 400 clients when running with two processors and 800 clients when running with four processors). We calculate a series of metrics representing the server behavior, and determine which of them are affected when increasing the number of clients. From these metrics, an in-depth analysis is performed looking for the causes of their dependence of server load.

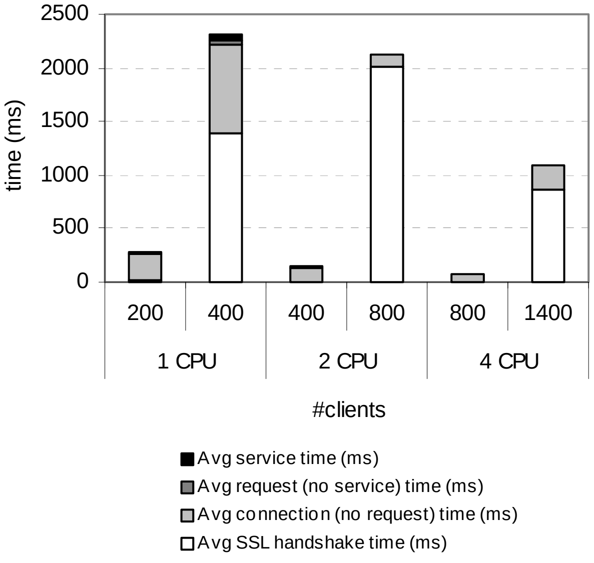

Figure 5. Average time spent by the server processing a persistent client connection

In order to detect the causes of server saturation we calculate, using the performance analysis framework, the average time spent by the server processing a persistent client connection, distinguishing the time devoted to each phase of the connection (connection phases have been described in Section 5.1) when running with different number of processors. This information is displayed in Figure 5. As shown in this figure, running with more processors decreases the average time required to process a connection. Notice that when the server is saturated, the average time required to handle a connection increases considerably. Going into detail on the connection phases, the time spent in the *SSL handshake* phase of the connection increases from 28 ms to 1389 ms when running with one processor, from 4 ms to 2003 ms when running with two processors and from 4 ms to 857 ms when running with four processors, becoming the phase where the server spends the major part of the time when processing a connection.

To determine the causes of the great increment of the time spent in the *SSL handshake* phase of the connection, we calculate the percentage of connections that perform a resumed SSL handshake (reusing the SSL Session ID) versus the percentage of connections that perform a full SSL handshake when running with different number of processors. This information is shown in Figure 6. Notice that when running with one processor and with 200 clients, the 97% of SSL handshakes can reuse the SSL connection, but with 400 clients, only the 27% can reuse it. The rest must negotiate the full SSL handshake, saturating the server because it cannot supply the computational demand of these full SSL handshakes. Remember the big difference between the computational demand of a resumed SSL handshake (2 ms) and a full SSL handshake (175 ms). The same situation is produced when running with two processors (the percentage of full SSL handshakes has increased from 0.25% to 68%), and when running with four processors (from 0.2% to 63%).

We have determined that when running with any number of processors the server saturates when most of the incoming client connections must negotiate a full SSL handshake instead of resuming an existing SSL connection, requiring a computing capacity that the available processors are unable to supply. Nevertheless, why does this occur from a given number of clients? In other words, why do incoming connections negotiate a full SSL handshake instead of a resumed SSL handshake when attending a given number of clients? Remember that we have configured the client with a timeout of 10 seconds. This means that if no reply is received in this time (the server is unable to supply it because it is heavy loaded), this client is discarded and a new one is initiated. Remember that the initiation of a new client requires the establishment of a new SSL connection, and therefore the negotiation of a full SSL handshake.

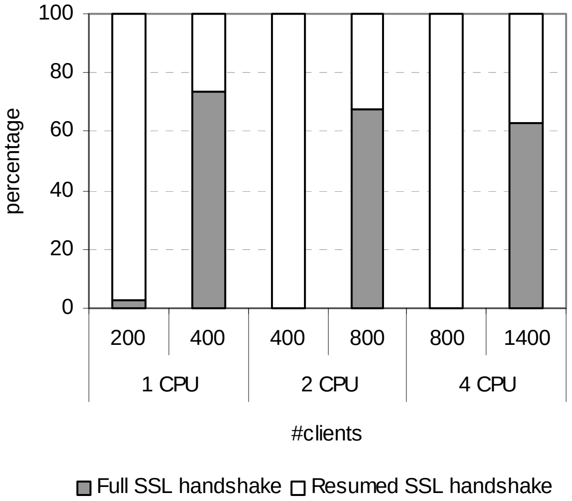

Figure 6. Incoming server connections classification depending on SSL handshake type performed

Therefore, if the server is loaded and it cannot handle the incoming requests before the client timeouts expire, this provokes the arrival of a great amount of new client connections that need the negotiation of a full SSL handshake, provoking the server performance degradation. This asseveration is supported with the information of Figure 7. This figure shows the number of clients timeouts occurred when running with different number of processors. Notice that from a given number of clients, the number of clients timeouts increases considerably, because the server is unable to respond to the clients before their timeouts expires. The comparison of this figure with Figure 4 reveals that this given number of clients matches with the saturation load of the server.

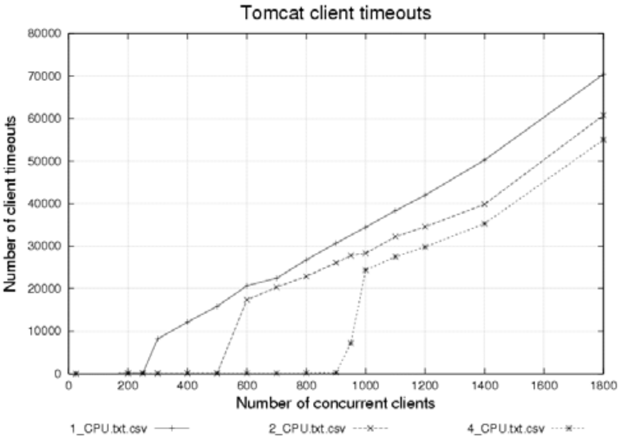

Figure 7. Client timeouts with different number of processors

In order to evaluate the effect on server of the great amount of full SSL handshakes, we calculate, using the performance analysis framework, the state of HttpProcessors when they are in the *SSL handshake* phase of the connection, which is shown in Figure 8. The HttpProcessors can be running (Run state), blocked waiting for the finalization of an input/output operation (Blocked I/O state), blocked waiting for the synchronization with other HttpProcessors in a monitor (Blocked Synch) or waiting for a free processor to become available to execute (Ready state). When the server is not saturated, HttpProcessors spent most of their time in Run state. But when the server is running with one processor and saturates (400 clients) HttpProcessors spent the 47% of their time in Ready state. This fact confirms that the server cannot handle all the incoming full SSL handshakes with only one processor.

It is expected that when the server is saturated and running with two or four processors, the HttpProcessors spent most part of their time of Ready state (waiting for a free processor to execute), in the same way as when running with one processor. But looking at Figure 8, we discover that when the server is running with two processors and saturates, although the time spent on Ready state has increased, the HttpProcessors spent the 70% of their time in Blocked Synch state (blocked waiting for the synchronization with other HttpProcessors in a monitor). This kind of contention can be produced due to the saturation of the available processors on multiprocessor systems, as occurred in this case. When running with four processors, the time spent in Ready state and Blocked Synch state is also increased.

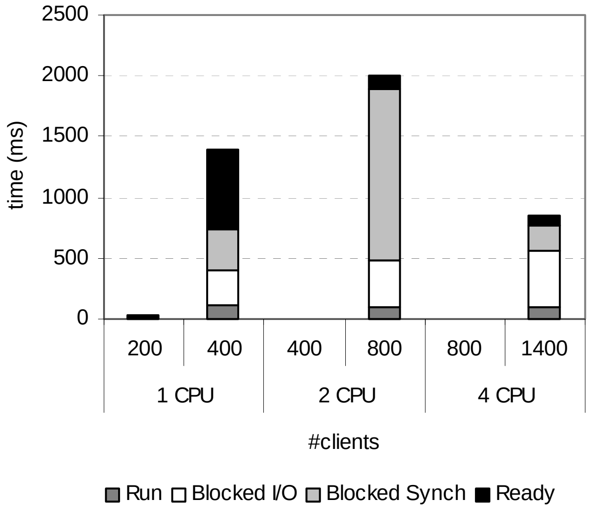

Figure 8. State of HttpProcessors when they are in the *SSL handshake* phase of a connection

Notice that, although the cause of the server saturation is the same when running with one, two or four processors (there are not processors enough to support demanded computation), this saturation is manifested in different forms (waiting for a processor to become available in order to execute or in a contention situation produced by the saturation of processors).

With the analysis performed, we can conclude that the processor is a bottleneck for Tomcat performance and scalability when running dynamic web applications in a secure environment. We have demonstrated that running with more processors makes the server able to handle more clients before saturating, and even when the server has reached a saturated state, better throughput can be obtained if running with more processors.

7. Related work

Application server scalability constitutes an important issue to support the increasing number of users of secure dynamic web sites. Although this work focuses on maintaining server scalability when running in secure environments adding more resources (vertical scaling), the great computational demand of SSL protocol can be handled using other approaches.

Major J2EE vendors such as BEA [6] or IBM [2][12] use clustering (horizontal scaling) to achieve scalability and high availability. Several studies evaluating server scalability using clustering have been performed [2][18], but none of them considers security issues.

Scalability can be also achieved delegating the security issues on a web server (e.g. Apache web server [4]) while the application server only processes dynamic web requests. In this case, the computational demand will be transferred to the web server, which can be optimized for SSL management.

It is also possible to add new specialized hardware for processing SSL requests [28], reducing the processor demand, but increasing the cost of the system.

Related with the vertical scalability covered in this paper, some works have evaluated this scalability on web servers or application servers. For example, [8] and [20] only consider static web content, and in [3][8][20][25] the evaluation is limited to a numerical study without performing an analysis to justify the scalability results obtained. Besides, none of these works evaluates the effect of security on application server scalability.

Other works try to improve application server scalability by tuning some server parameters and/or JVM options and/or operating system properties. For example, Tomcat scalability while tuning some parameters, including different JVM implementations, JVM flags and XML implementations has been studied in [24]. In the same way, the application server scalability using different mechanisms for generating dynamic web content has been evaluated in [11]. However, none of these works considers any kind of scalability relative to resources (neither vertical nor horizontal), neither the influence of security on the application server scalability.

Certain kind of analysis has been performed in some works. For example, [1] and [11] provide a quantitative analysis based on general metrics of application server execution collecting system utilization statistics (CPU, memory, network bandwidth, etc.). These statistics may allow the detection of some application server bottlenecks, but this coarse-grain analysis is often not enough when dealing with more sophisticated performance problems.

The influence of security on application server scalability has been covered in some works. For example, the performance and architectural impact of SSL on the servers in terms of various parameters such as throughput, utilization, cache sizes and cache miss ratios has been analyzed in [23], concluding that SSL increases computational cost of transactions by a factor of 5-7. The impact of each individual operation of TLS protocol in the context of web servers has been studied in [13], showing that key exchange is the slowest operation in the protocol. [16] analyzes the impact of full handshake in connection establishment and proposes caching sessions to reduce it.

Security for Web Services can be also provided with SSL, but other proposals as WS-Security [21], which uses industry standards like XML Encryption and XML Signature, have been made. Coupled with WS-SecureConversation, the advantage WS-Security has over SSL over HTTP is twofold: first, it works independently of the underlying transport protocol and second, it provides security mechanisms that operate in end-to-end scenarios (across trust boundaries) as opposed to point-to-point scenarios (i.e. SSL). Anyway, WS-Security requires also a great computational demand to support the encryption mechanisms, making most of the conclusions obtained in this paper valid in Web Services environments too.

Our approach intends to achieve a complete characterization of dynamic web applications using SSL vertical scalability determining the causes of server saturation performing a detailed analysis of application server behavior considering all levels involved in the execution of dynamic web applications.

8. Conclusions

In this paper we have presented a complete characterization of Tomcat application server scalability when executing the RUBiS benchmark using SSL, which is very valuable considering the few related work in this topic. This characterization is divided in two parts.

First, we have measured Tomcat vertical scalability (i.e. adding more processors) when using SSL and we have analyzed the effect of this addition on server scalability. The results confirmed that running with more processors makes the server able to handle more clients before saturating and even when the server has reached a saturated state, better throughput can be obtained if running with more processors. Second, we have analyzed the causes of server saturation when running with different number of processors using a performance analysis framework. This framework allows a fine-grain analysis of dynamic web applications by considering all levels involved in their execution. Our analysis has revealed that the processor is a bottleneck for Tomcat performance on secure environments and could make sense to upgrade the system adding more processors to improve the server scalability.

The results obtained in this work demonstrate the convenience of incorporating to the Tomcat server some kind of overload control mechanism to avoid the throughput degradation produced due to the massive arrival of full SSL connections that we have shown in this paper. The server could differentiate full SSL connections from resumed SSL connections limiting the acceptation of full SSL connections to the maximum number acceptable without saturating, while accepting all the resumed SSL connections to maximize the number of client sessions successfully completed.

Although this work focuses on maintaining server scalability when running in secure environments adding more resources (vertical scalability), the great computational demand of SSL protocol can be handled using other approaches as described in Section 7. Whichever approach is used, it is necessary to consider the security as an important issue that can heavily affect the scalability and performance of web applications.

9. Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIC2001-0995-C02-01 and TIN2004-07739-C02-01 and by the CEPBA (European Center for Parallelism of Barcelona). For additional information about the authors, please visit the Barcelona eDragon Research Group web site [5].

10. References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. *Specification and Implementation of Dynamic Web Site Benchmarks*. IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA. November 25, 2002.

[2] Y. An, T. K. T. Lau and P. Shum. A Scalability Study for WebSphere Application Server and DB2. IBM white paper. January 2002. http://www-106.ibm.com/developerworks/db2/library/techarticle/0202an/0202an.pdf

[3] S. Anne, A. Dickson, D. Eaton, J. Guizan and R. Maiolini. JBoss 3.2.1 vs. WebSphere 5.0.2 Trade3 Benchmark. SMP Scaling: Comparison report. SWG Competitive Technology Lab. October 2003. http://www.werner.be/blog/resources/werner/JBoss_3.2.1_vs_WAS_5.0.2.pdf

[4] Apache HTTP Server Project
http://httpd.apache.org/

[5] Barcelona eDragon Research Group
http://www.cepba.upc.es/eDragon

[6] BEA Systems, Inc. Achieving Scalability and High Availability for E-Business. BEA white paper. March 2003.
http://dev2dev.bea.com/products/wlserver81/whitepapers/WLS_81_Clustering.jsp

[7] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. SIGMETRICS'98, pp. 151-160, Madison, Wisconsin, USA. June 24-26, 1998.

[8] V. Beltran, D. Carrera, J. Torres and E. Ayguade. Evaluating the Scalability of Java Event-Driven Web Servers. 2004 International Conference on Parallel Processing (ICPP'04), pp. 134-142, Montreal, Canada. August 15-18, 2004.

[9] V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Performance Impact of Using SSL on Dynamic Web Applications. XV Jornadas de Paralelismo, pp. 471-476, Almeria, Spain. September 15-17, 2004. http://people.ac.upc.es/jguitart/HomepageFiles/Jornadas04.pdf

[10] D. Carrera, J. Guitart, J. Torres, E. Ayguade and J. Labarta. Complete Instrumentation Requirements for Performance Analysis of Web based Technologies. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), pp. 166-176, Austin, Texas, USA. March 6-8, 2003.

[11] E. Cecchet, J. Marguerite and W. Zwaenepoel. Performance and Scalability of EJB Applications. 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pp. 246-261. Seattle, Washington, USA. November 4-8, 2002

[12] W. Chiu. Design for Scalability. IBM white paper. September 2001. http://www-106.ibm.com/developerworks/websphere/library/techarticles/hvws/scalability.html

[13] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. 9th Network and Distributed System Security Symposium (NDSS'02), San Diego, California, USA. February 6-8, 2002.

[14] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246. January 1999.

[15] A. O. Freier, P. Karlton, and C. Kocher. The SSL Protocol, Version 3.0. November 1996.

[16] A. Goldberg, R. Buff and A. Schmitt. Secure Web Server Performance Dramatically Improved by Caching SSL Session Keys. Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), Madison, Wisconsin, USA. June 23, 1998.

[17] J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Tuning Dynamic Web Applications using Fine-Grain Analysis. 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05), Lugano, Switzerland. February 9-11, 2005.

[18] I. Haddad and G. Butler. Experimental Studies of Scalability in Clustered Web System. Workshop on Communication Architecture for Clusters (CAC'04) (in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'04)), Santa Fe, New Mexico, USA. April 26, 2004.

[19] I. Haddad. Scalability Issues and Clustered Web Servers. Technical Report. Concordia University. August 13, 2000.

[20] I. Haddad. Open-Source Web Servers: Performance on Carrier-Class Linux Platform. Linux Journal, Volume 2001, Issue 91, page 1. November 2001.

[21] IBM Corporation, Microsoft Corporation and VeriSign Inc. Web Services Security (WS-Security) Specification. Version 1.0.05. April 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/

[22] Jakarta Tomcat Servlet Container
http://jakarta.apache.org/tomcat

[23] K. Kant, R. Iyer, and P. Mohapatra. Architectural Impact of Secure Socket Layer on Internet Servers. 2000 IEEE International Conference on Computer Design (ICCD'00), pp. 7-14, Austin, Texas, USA. September 17-20, 2000.

[24] P. Lin. So You Want High Performance (Tomcat Performance). September 2003. http://jakarta.apache.org/tomcat/articles/performance.pdf

[25] M. Malzacher and T. Kochie. Using a Web application server to provide flexible and scalable e-business solutions. IBM white paper. April 2002. http://www-900.ibm.com/cn/software/websphere/products/download/whitepapers/performance_40.pdf

[26] Microsoft Active Server Pages
http://www.asp.net

[27] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), pp. 59-67. Madison, Wisconsin, USA. June 23, 1998.

[28] R. Mraz. SecureBlue: An Architecture for a High Volume SSL Internet Server. 17th Annual Computer Security Applications Conference (ACSAC'01), New Orleans, Louisiana, USA. December 10-14, 2001.

[29] MySQL
http://www.mysql.com

[30] Paraver
http://www.cepba.upc.es/paraver

[31] PHP Hypertext Preprocessor
http://www.php.net

[32] E. Rescorla. HTTP over TLS. RFC 2818. May 2000.

[33] Sun Microsystems. Enterprise Java Beans Technology
http://java.sun.com/products/ejb

[34] Sun Microsystems. Java Servlets Technology
http://java.sun.com/products/servlet